Your first program

In this tutorial, we will write a simple Factor program which prompts the user to enter a word, and tests if it is a palindrome (that is, the word is spelled the same backwards and forwards).

In this tutorial, you will learn about basic Factor development tools.

Creating a vocabulary for your first program

Factor source code is organized into **Vocabularies**. Before we can write our first program, we must create a vocabulary for it.

Start by loading the scaffold tool:

```
USE: tools.scaffold
```

Then, ask the scaffold tool to create a new vocabulary named `palindrome`:

```
"resource:work" "palindrome" scaffold-vocab
```

If you look at the output, you will see that a few files were created in your "work" directory, and that the new source file was loaded.

The following phrase will print the full path of your work directory:

```
"work" resource-path .
```

The work directory is one of several **Vocabulary roots** where Factor searches for vocabularies. It is possible to define new vocabulary roots; see **Working with code outside of the Factor source tree**. To keep things simple in this tutorial, we'll just use the work directory, though.

Open the work directory in your file manager, and open the subdirectory named `palindrome`. Inside this subdirectory you will see a file named `palindrome.factor`. Open this file in your text editor.

You are now ready to go on to the next section: **Writing some logic in your first program**.

Writing some logic in your first program

The Factor workflow is to edit source code on disk and then to refresh the live image. Let's examine the file that we just created with the scaffold tool.

Your `palindrome.factor` file should look like the following after the previous section:

```
! Copyright (C) 2011 <your name here>
! See http://factorcode.org/license.txt for BSD license.
USING: ;
IN: palindrome
```

Notice that the file ends with an **IN:** form telling Factor that all definitions in this source file should go into the `palindrome` vocabulary using the **IN:** word. We will add new definitions after the **IN:** form.

In order to be able to call the words defined in the `palindrome` vocabulary, you need to issue the following command in the listener:

```
USE: palindrome
```

Now, we will be making some additions to the file. Since the file was loaded by the scaffold tool in the previous step, you need to tell Factor to reload it if it changes. Factor has a handy feature for this; pressing `F2` in the listener window will reload any changed source files. You can also force a single vocabulary to reload, in case the refresh feature does not pick up changes from disk:

```
"palindrome" reload
```

We will now write our first word using **:**. This word will test if a string is a palindrome; it will take a string as input, and give back a boolean as output. We will call this word `palindrome?` , following a naming convention that words returning booleans have names ending with `?`.

Recall that a string is a palindrome if it is spelled the same forwards or backwards; that is, if the string is equal to its reverse. We can express this in Factor as follows:

```
: palindrome? ( string -- ? ) dup reverse = ;
```

Place this definition at the end of your source file.

Now we have changed the source file, we must reload it into Factor so that we can test the new definition. To do this, simply go to the Factor listener and press `F2`. This will find any previously-loaded source files which have changed on disk, and reload them.

When you do this, you will get an error about the **dup** word not being found. This is because this word is part of the **kernel** vocabulary, but this vocabulary is not part of the source file's **Parse-time word lookup**. You must explicitly list dependencies in source files. This allows Factor to automatically load required vocabularies and makes larger programs easier to maintain.

To add the word to the search path, first convince yourself that this word is in the **kernel** vocabulary. Enter `dup` in the listener's input area, and press `H`. This will open the documentation browser tool, viewing the help for the **dup** word. One of the subheadings in the help article will mention the word's vocabulary.

Go back to the third line in your source file and change it to:

```
USING: kernel ;
```

Next, find out what vocabulary **reverse** lives in; type the word name `reverse` in the listener's input area, and press `H`.

It lives in the **sequences** vocabulary, so we add that to the search path:

```
USING: kernel sequences ;
```

Finally, check what vocabulary **=** lives in, and confirm that it's in the **kernel** vocabulary, which we've already added to the search path.

Now press F2 again, and the source file should reload without any errors. You can now go on and learn about **Testing your first program**.

Testing your first program

Your `palindrome.factor` file should look like the following after the previous section:

```
! Copyright (C) 2011 <your name here>
! See http://factorcode.org/license.txt for BSD license.
USING: kernel sequences ;
IN: palindrome

: palindrome? ( str -- ? ) dup reverse = ;
```

We will now test our new word in the listener. If you haven't done so already, add the palindrome vocabulary to the listener's vocabulary search path:

```
USE: palindrome
```

Next, push a string on the stack:

```
"hello"
```

Note that the stack display in the listener now shows this string. Having supplied the input, we call our word:

```
palindrome?
```

The stack display should now have a boolean false - **f** - which is the word's output. Since "hello" is not a palindrome, this is what we expect. We can get rid of this boolean by calling **drop**. The stack should be empty after this is done.

Now, let's try it with a palindrome; we will push the string and call the word in the same line of code:

```
"racecar" palindrome?
```

The stack should now contain a boolean true - **t**. We can print it and drop it using the **.** word:

```
.
```

What we just did is called *interactive testing*. A more advanced technique which comes into play with larger programs is **Unit testing**.

Create a test harness file using the scaffold tool:

```
"palindrome" scaffold-tests
```

Now, open the file named `palindrome-tests.factor`; it is located in the same directory as `palindrome.factor`, and it was created by the scaffold tool.

We will add some unit tests, which are similar to the interactive tests we did above. Unit tests are defined with the **unit-test** word, which takes a sequence of expected outputs, and a piece of code. It runs the code, and asserts that it outputs the expected values.

Add the following two lines to `palindrome-tests.factor`:

```
[ f ] [ "hello" palindrome? ] unit-test
[ t ] [ "racecar" palindrome? ] unit-test
```

Now, you can run unit tests:

```
"palindrome" test
```

It should report that all tests have passed. Now you can read about **Extending your first program**.

Extending your first program

Our palindrome program works well, however we'd like to extend it to ignore spaces and non-alphabetical characters in the input.

For example, we'd like it to identify the following as a palindrome:

```
"A man, a plan, a canal: Panama."
```

However, right now, the simplistic algorithm we use says this is not a palindrome:

```
"A man, a plan, a canal: Panama." palindrome? .
f
```

We would like it to output **t** there. We can encode this requirement with a unit test that we add to `palindrome-tests.factor`:

```
[ t ] [ "A man, a plan, a canal: Panama." palindrome? ] unit-test
```

If you now run unit tests, you will see a unit test failure:

```
"palindrome" test
```

The next step is to, of course, fix our code so that the unit test can pass.

We begin by writing a word which removes blanks and non-alphabetical characters from a string, and then converts the string to lower case. We call this word `normalize`. To figure out how to write this word, we begin with some interactive experimentation in the listener.

Start by pushing a character on the stack; notice that characters are really just integers:

```
CHAR: a
```

Now, use the **Letter?** word to test if it is an alphabetical character, upper or lower case:

```
Letter? .
t
```

This gives the expected result.

Now try with a non-alphabetical character:

```
CHAR: #

Letter? .
f
```

What we want to do is given a string, remove all characters which do not match the **Letter?** predicate. Let's push a string on the stack:

```
"A man, a plan, a canal: Panama."
```

Now, place a quotation containing **Letter?** on the stack; quoting code places it on the stack instead of executing it immediately:

```
[ Letter? ]
```

Finally, pass the string and the quotation to the **filter** word:

```
filter
```

Now the stack should contain the following string:"AmanaplanacanalPanama". This is almost what we want; we just need to convert the string to lower case now. This can be done by calling **>lower**; the `>` prefix is a naming convention for conversion operations, and should be read as "to":

```
>lower
```

Finally, let's print the top of the stack and discard it:

```
.
```

This will output `amanaplanacanalpanama`. This string is in the form that we want, and we evaluated the following code to get it into this form:

```
[ Letter? ] filter >lower
```

This code starts with a string on the stack, removes non-alphabetical characters, and converts the result to lower case, leaving a new string on the stack. We put this code in a new word, and add the new word to `palindrome.factor`:

```
: normalize ( str -- newstr ) [ Letter? ] filter >lower ;
```

You will need to add **unicode.case** and **unicode.categories** to the vocabulary search path, so that **>lower** and **Letter?** can be used in the source file.

We modify `palindrome?` to first apply `normalize` to its input:

```
: palindrome? ( str -- ? ) normalize dup reverse = ;
```

Factor compiles the file from the top down. So, be sure to place the definition for `normalize` above the definition for `palindrome?`.

Now if you press `F2`, the source file should reload without any errors. You can run unit tests again, and this time, they will all pass:

```
"palindrome" test
```